Tecniche Algoritmiche/1
Divide et Impera

Gianluigi Zavattaro
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna
gianluigi.zavattaro@unibo.it

Slide realizzate a partire da materiale fornito dal Prof. Moreno Marzolla

Tecniche algoritmiche

- Divide-et-impera
  - Un problema viene suddiviso in sotto-problemi, che vengono risolti ricorsivamente (top-down)
- Algoritmi greedy
  - Ad ogni passo si fa sempre la scelta che in quel momento appare ottima; le scelte fatte non vengono mai disfatte
- Programmazione dinamica
  - La soluzione viene costruita (bottom-up) a partire da un insieme di sotto-problemi

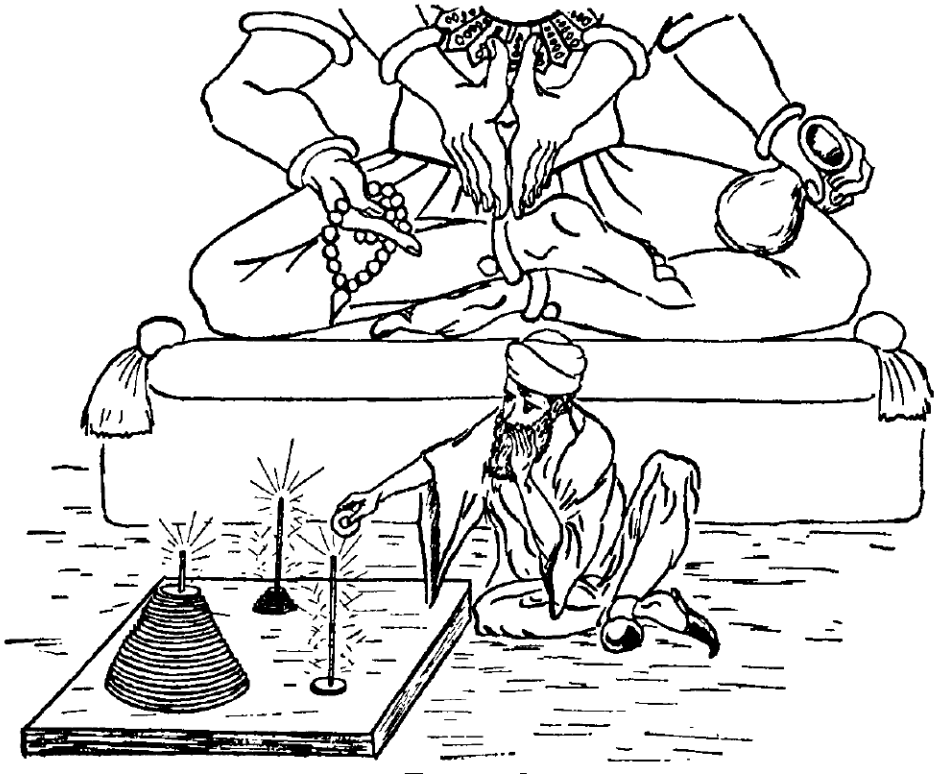
Divide-et-impera

- Tre fasi:
  - *Divide*: Dividi il problema in sotto-problemi indipendenti, di dimensioni “minori”
  - *Impera*: Risolvi i sotto-problemi ricorsivamente
  - *Combina*: Unisci le soluzioni dei sottoproblemi per costruire la soluzione del problema di partenza
- Non esiste una “ricetta” unica per implementare un algoritmo divide-et-impera:
  - Ricerca binaria: “divide” banale, niente fase di “combina”
  - Quick Sort: “divide” complesso, niente fase di “combina”
  - Merge Sort: “divide” banale, “combina” complesso

Le torri di Hanoi

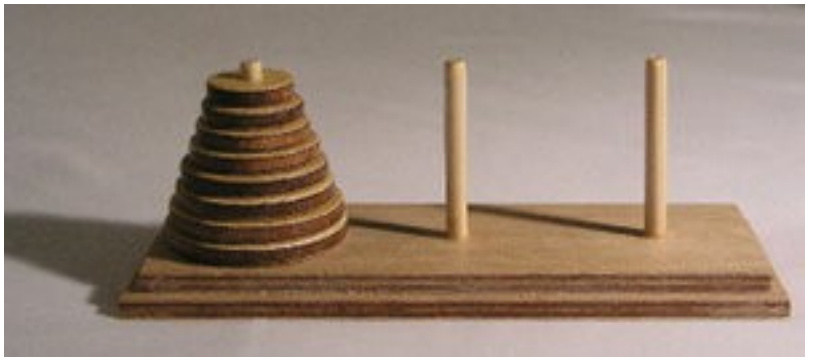
Le torri di Hanoi

Narra la leggenda che in un tempio indiano si trovi una stanza contenente una lastra di bronzo con sopra tre pioli di diamante. Dopo la creazione dell'universo, 64 dischi d'oro sono stati infilati in uno dei pioli in ordine decrescente di diametro, con il disco più largo appoggiato al piano di bronzo. Da allora, i monaci del tempio si alternano per spostare i dischi dal piolo originario in un altro piolo, seguendo le rigide regole di Brahma: i dischi vanno maneggiati uno alla volta, e non si deve mai verificare che un disco più largo sovrasta uno più piccolo sullo stesso piolo. Quando tutti i dischi saranno spostati su un altro piolo, l'universo avrà fine.



Fonte: George Gamow, *One, two, tree... infinity!*, Dover Publications, 1947

Le torri di Hanoi



http://en.wikipedia.org/wiki/Tower_of_Hanoi

- Gioco matematico
  - tre pioli
  - *n* dischi di dimensioni diverse
  - Inizialmente tutti i dischi sono impilati in ordine decrescente (più piccolo in alto) nel piolo di sinistra
- Scopo del gioco
  - Impilare in ordine decrescente i dischi sul piolo di destra
  - Senza mai impilare un disco più grande su uno più piccolo
  - Muovendo un disco alla volta
  - Utilizzando se serve anche il piolo centrale

Le torri di Hanoi
Soluzione divide-et-impera

http://en.wikipedia.org/wiki/Tower_of_Hanoi

```
Hanoi(Stack p1, Stack p2, Stack p3, integer n)
  if (n = 1) then
    p3.push(p1.pop())
  else
    hanoi(p1, p3, p2, n-1)
    p3.push(p1.pop())
    hanoi(p2, p1, p3, n-1)
  endif
```

Sposta *n* dischi da p1 a p3
usando p2 come appoggio

- Divide:
  - *n* - 1 dischi da p1 a p2
  - 1 disco da p1 a p3
  - *n* - 1 dischi da p2 a p3
- Impera
  - Esegui ricorsivamente gli spostamenti

Le torri di Hanoi
Soluzione divide-et-impera

```
Hanoi(Stack p1, Stack p2, Stack p3, integer n)
  if (n = 1) then
    p3.push(p1.pop())
  else
    hanoi(p1, p3, p2, n-1)
    p3.push(p1.pop())
    hanoi(p2, p1, p3, n-1)
  endif
```

- Costo computazionale:
  - $T(1) = 1$
  - $T(n) = 2\,T(n-1) + 1$ per $n > 1$
- **Domanda:** Quale è la soluzione della ricorrenza?
  - Se i monaci effettuano una mossa al secondo, per trasferire tutti i 64 dischi servirebbe un tempo pari a circa 127 volte l'età del nostro sole

Moltiplicazione di interi

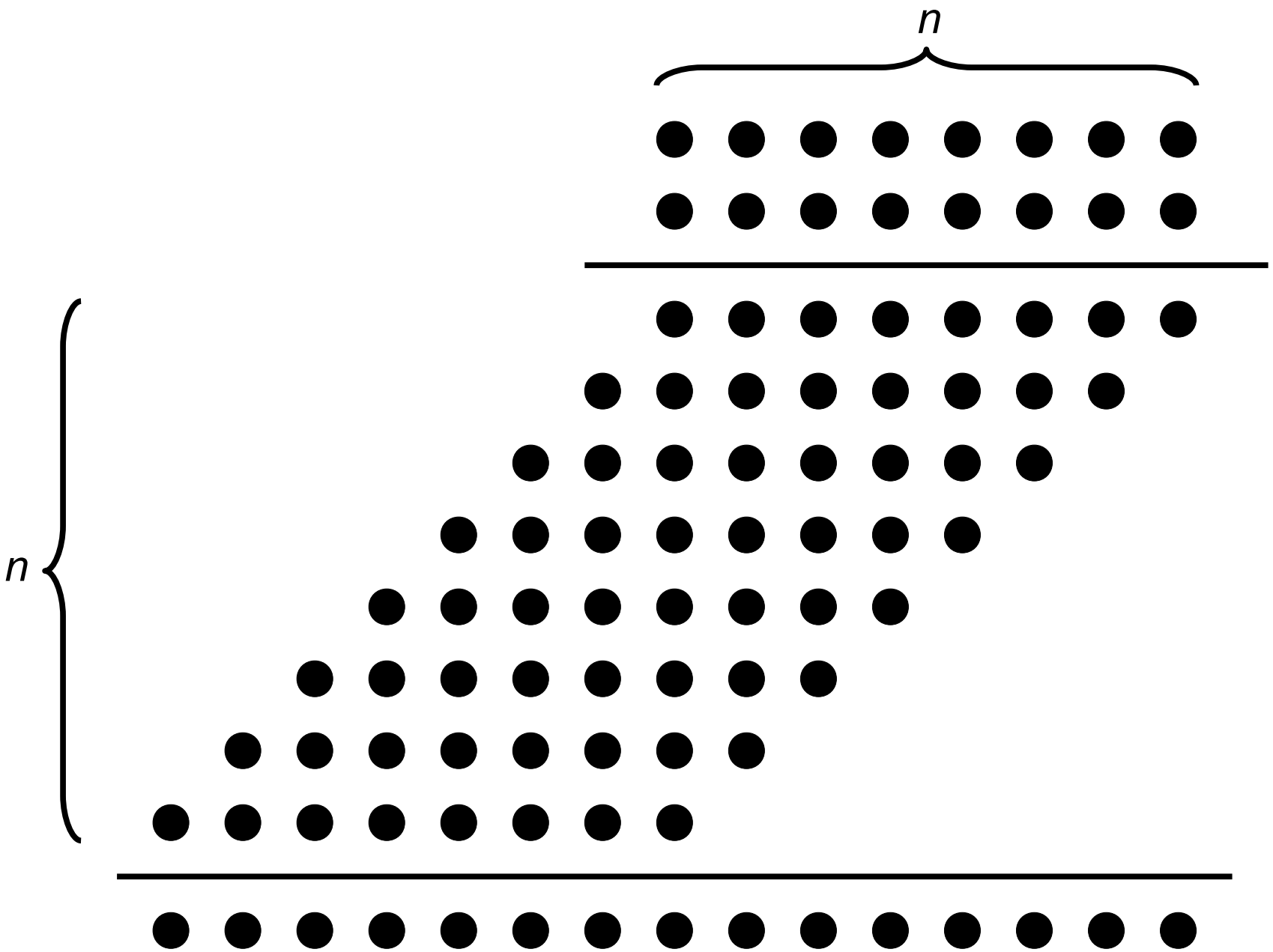
Moltiplicazione di interi di grandezza arbitraria

- Consideriamo due interi di *n* cifre decimali, *X* e *Y*

$$X = x_{n-1}\, x_{n-2} \ldots x_1\, x_0 = \sum_{i=0}^{n-1} x_i \times 10^i$$

$$Y = y_{n-1}\, y_{n-2} \cdots y_1\, y_0 = \sum_{i=0}^{n-1} y_i \times 10^i$$

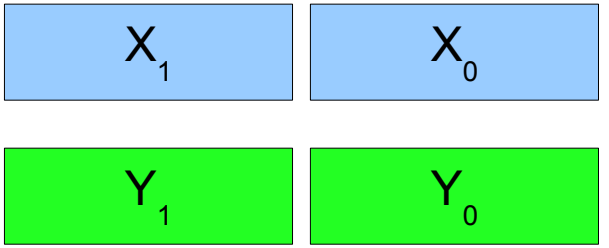
- Vogliamo calcolare il prodotto *XY*
  - L'algoritmo “moltiplicazione in colonna” (vedi prossima slide) che abbiamo imparato a scuola ha costo $O(n^2)$
  - Proviamo a fare di meglio con un algoritmo di tipo divide et impera


n
n

Idea

- Supponiamo che sia *X* che *Y* abbiano lo stesso numero di cifre (possiamo aggiungere zeri all'inizio)
- Dividiamo le sequenze di cifre in due parti uguali

$$X = X_1 \times 10^{n/2} + X_0$$
$$Y = Y_1 \times 10^{n/2} + Y_0$$



- Possiamo quindi calcolare il prodotto come:

$$X \times Y = (X_1 10^{n/2} + X_0) \times (Y_1 10^{n/2} + Y_0)$$
$$= (X_1 Y_1) \times 10^n + (X_1 Y_0 + X_0 Y_1) \times 10^{n/2} + X_0 Y_0$$

Osservazione

$$X \times Y = (X_1 Y_1) \times 10^n + (X_1 Y_0 + X_0 Y_1) \times 10^{n/2} + X_0 Y_0$$

- La moltiplicazione per 10^n richiede tempo O(n)
  - Equivale ad uno shift a sinistra di n posizioni
- Ci sono 4 prodotti di numeri a $n/2$ cifre
- Possiamo scrivere la relazione di ricorrenza

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 4\,T(n/2) + c_2 n & \text{altrimenti} \end{cases}$$

- Soluzione (Master Theorem, caso 1): O(n^2)
  - Non abbiamo migliorato nulla rispetto all'algoritmo banale :-(

Miglioramento

- Se poniamo

$$P_1 = (X_1 + X_0) \times (Y_1 + Y_0)$$
$$P_2 = (X_1 Y_1)$$
$$P_3 = (X_0 Y_0)$$

  possiamo scrivere

$$X \times Y = P_2 10^n + (P_1 - P_2 - P_3) \times 10^{n/2} + P_3$$

- Il calcolo di P1, P2 e P3 richiede in tutto solo 3 prodotti tra numeri di *n*/2 cifre

Analisi del miglioramento

- Otteniamo la nuova relazione di ricorrenza

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 3\,T(n/2) + c_2 n & \text{altrimenti} \end{cases}$$

- In base al Master Theorem, essendo a = 3, b = 2, e quindi α = log 3/log 2 = $\log_2 3$, avremo (per caso 1):

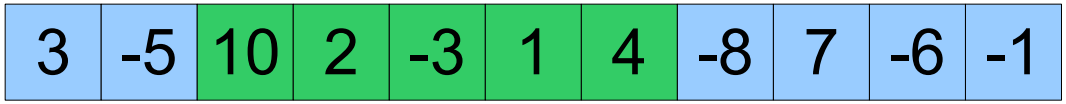
$$T(n) \in \mathrm{O}\left(n^{\log_2 3}\right) \approx \mathrm{O}\left(n^{1.59}\right)$$

Sottovettore non vuoto di valore massimo

Sottovettore di valore massimo

- Vettore $v[1..n]$ di n valori reali arbitrari
- Vogliamo individuare un sottovettore non vuoto di v la somma dei cui elementi sia massima

Elementi contigui!



- Domanda: quanti sono i sottovettori di v?
  - 1 sottovettore di lunghezza n
  - 2 sottovettori di lunghezza n - 1
  - 3 sottovettori di lunghezza n - 2
  - ...
  - k sottovettori di lunghezza $n - k + 1$
  - ...
  - n sottovettori di lunghezza 1

Risposta:
$n(n+1)/2 \in O(n^2)$ sottovettori

Prima versione

```
double SommaMax1( double v[1..n] )
    double smax ← v[1];
    for integer i ← 1 to n do
        for integer j ← i to n do
            double s ← 0;
            for integer k ← i to j do
                s ← s + v[k];
            endfor
            if (s > smax) then
                smax ← s;
            endif
        endfor
    endfor
    return smax;
```

Costo?

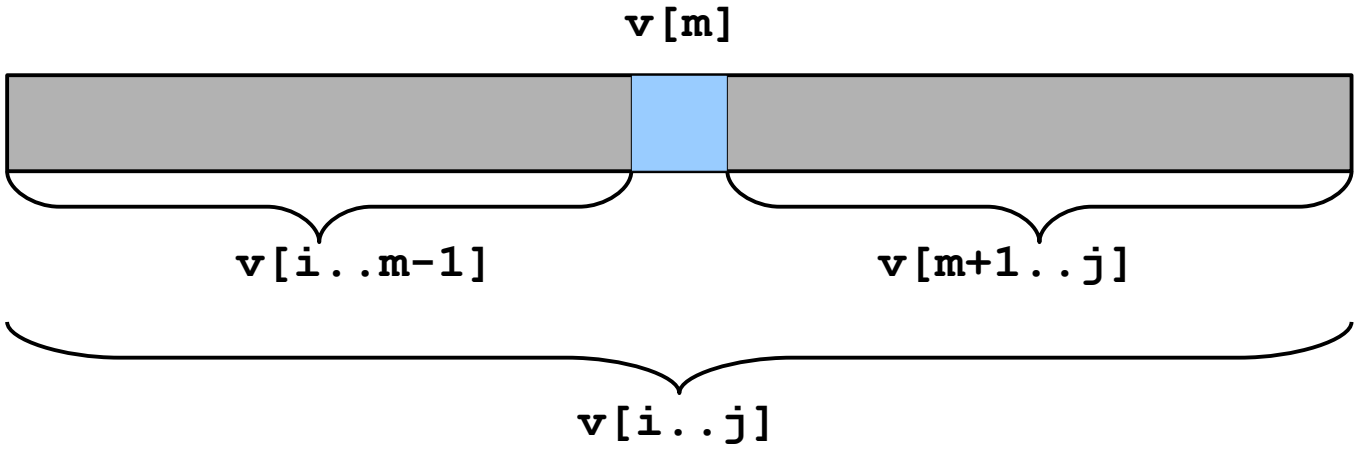
Seconda versione

```
double SommaMax2( double v[1..n] )
    double smax ← v[1];
    for integer i ← 1 to n do
        double s ← 0;
        for integer j ← i to n do
            s ← s + v[j];
            if (s > smax) then
                smax ← s;
            endif
        endfor
    endfor
    return smax;
```

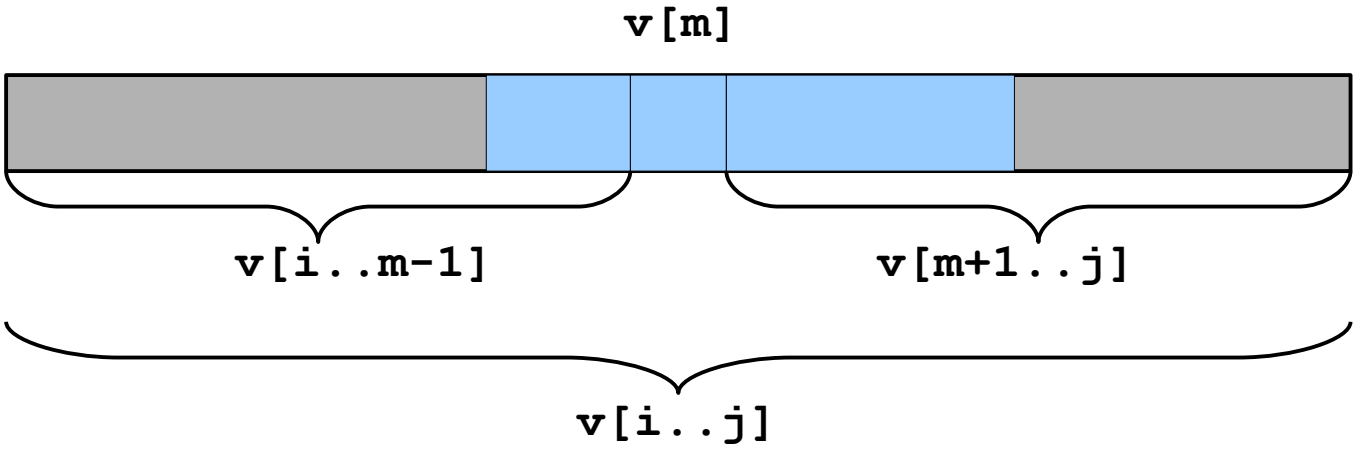
Costo?

Strategia divide-et-impera

- Dividiamo il vettore in due parti, separate dall'elemento in posizione centrale **`v[m]`**
- Il sottovettore di somma massima potrebbe trovarsi:
  - Interamente nella prima metà **`v[i..m-1]`**
  - Interamente nella seconda metà **`v[m+1..j]`**
  - "A cavallo" tra la prima e la seconda metà

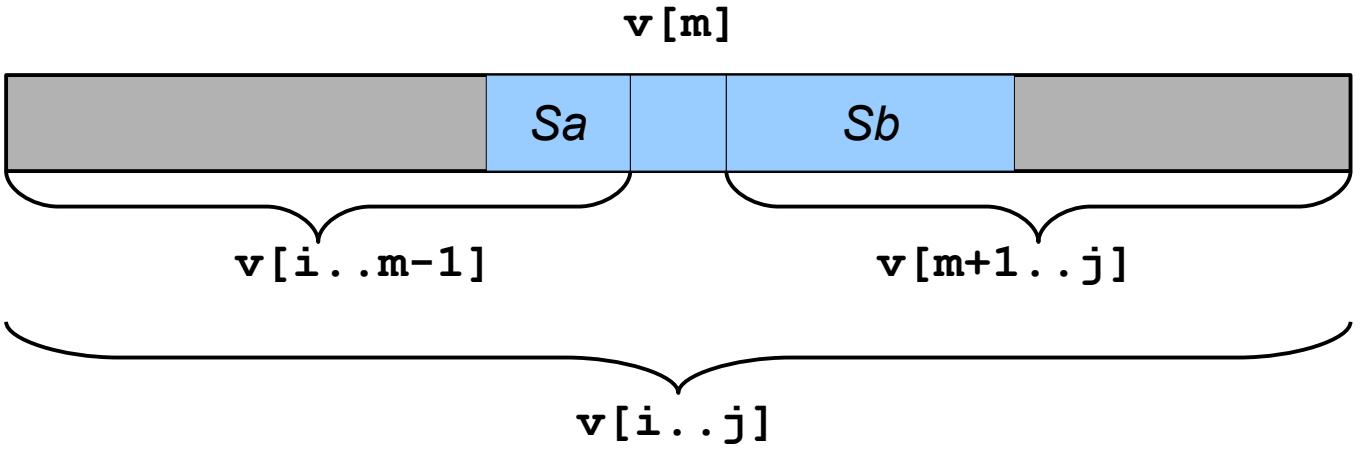


Strategia divide-et-impera



- La parte più complicata è la ricerca del sottovettore di somma massima che contiene **`v[m]`**
- Tale vettore
  - Puo' includere una parte prima di **`v[m]`**
  - Puo' includere una parte dopo di **`v[m]`**

Strategia divide-et-impera



- Calcolo la max somma *Sa* tra tutti i sottovettori (incluso quello vuoto) il cui ultimo elemento è **`v[m-1]`**
- Calcolo la max somma *Sb* tra tutti i sottovettori (incluso quello vuoto) il cui primo elemento è **`v[m+1]`**
- Il sottovettore di somma max che include **`v[m]`** avrà somma (v[m] + Sa + Sb)

Terza versione basata su divide-et-impera

```
real SommaMaxDI( real V[1..n], integer i, j )
    if (i>j) return 0
    else if (i==j) return V[i]
    else
        m ← floor((i+j)/2);
        real l ← SommaMaxDI( V, i, m-1 )
        real r ← SommaMaxDI( V, m+1, j )
        real sa ← 0, sb ← 0, s ← 0;
        integer k;
        for (k ← m-1; k>=i; k--) {
            s ← s + V[k];
            if (s > sa) sa ← s;
        }
        s ← 0;
        for (k ← m+1; k<=j; k++) {
            s ← s + V[k];
            if (s > sb) sb ← s;
        }
        return max(l, r, V[m]+sa+sb);
```

Costo?

Sottovettore di valore massimo

- Abbiamo visto tre possibili soluzioni:
  - Forza bruta:
    controllando separatamente tutti i sottovettori – costo $O(n^3)$
  - Versione migliorata:

    una volta calcolato il valore di un sottovettore, lo usiamo per calcolare il valore del sottovettore con un ulteriore elemento in più – costo $O(n^2)$
  - Divide-et-impera:

    si cercano i sottovettori di valore massimo separatamente nelle due metà, e si confrontano con il massimo sottovettore che include il valor medio – costo $O(n \log n)$

Metodo divide-et-impera

- Quando applicare divide-et-impera
  - I passi “divide” e “combina” devono essere “semplici”
  - Idealmente, il costo complessivo deve essere migliore del corrispondente algoritmo iterativo

Esercizio 1

- Consideriamo un array A[1..n] composto da n ≥ 0 valori reali, non necessariamente distinti.
  L'array è ordinato in senso non decrescente.
  Scrivere un algoritmo ricorsivo di tipo divide-et-impera che restituisca true se e solo se A contiene valori duplicati.
  Calcolare il costo computazionale dell'algoritmo proposto.

Esercizio 2

- Scrivere un algoritmo ricorsivo di tipo divide-et-impera che, dato un array A[1..n] di valori reali, restituisce true se e solo se A è ordinato in senso non decrescente, cioè se $A[1] \leq A[2] \leq \ldots \leq A[n]$. Calcolare il costo computazionale dell'algoritmo proposto.

Esercizio 3

- Si consideri un array A[1..n] composto da n ≥ 1 interi distinti ordinati in senso crescente (cioè A[1] < A[2] < … < A[n]).
  Scrivere un algoritmo efficiente che, dato in input l'array A, determina un indice i, se esiste, tale che A[i] = i. Nel caso esistano più indici che soddisfano la relazione precedente, è sufficiente restituirne uno qualsiasi.
  Determinare il costo computazionale dell'algoritmo.

Esercizio 4

- Si consideri un array A[1..n] contenente valori reali ordinati in senso non decrescente; l'array può contenere valori duplicati.
  Scrivere un algoritmo ricorsivo di tipo divide-et-impera che, dato A e due valori reali low < up, calcola quanti valori di A appartengono all'intervallo [low, up].
  Determinare il costo computazionale dell'algoritmo proposto.